



International Conference on Computational Science, ICCS 2012

A Study of Performance Portability Using Piecewise-Parabolic Method (PPM) Gas Dynamics Applications

Pei-Hung Lin^{a,b*}, Jagan Jayaraj^{a,b}, Paul Woodward^a, Pen-Chung Yew^b^aLaboratory for Computational Science & Engineering, University of Minnesota, MN 55455, USA^bDepartment of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455, USA

Abstract

The past decade has produced numerous CPU architectural innovations. These have included multiple cores per CPU, multiple simultaneous threads per core, and, especially with GPUs, highly complex memory hierarchies. As a result, performance portability has become a major challenge to programmers. We identify the SIMD engines in modern CPU and GPU cores as the key to obtaining high performance for scientific application codes. This common element of all present computing devices makes performance portability possible. However, we find that achieving this performance requires us to express the code in terms of intrinsic functions for the SIMD engine instructions, and these functions are different for each device. To assist the programmer in creating the necessary code expressions for each vendor's compilers, we have built an automated code translator that takes as input a single Fortran source code, written in a special style and annotated with directives, and creates output code for each device and compiler combination. The manual translations for GPU permit us here to evaluate the performance that our code transformations deliver on these devices. We present a performance study using our single-fluid PPM gas dynamics code and covering the latest multi-core processors and the Nvidia GPU.

Keywords: high-performance computing; optimization; scientific computation; CFD; parallel computing; GPGPU

1. Introduction

Recent innovations in computer architecture design have resulted in challenges in application portability. Language extensions and new programming models have been developed to accommodate these changes in hardware design. However, to apply these new programming features and keep the same application running on multiple platforms, programmers must make significant new efforts. Tuning the applications to achieve performance on multiple platforms is a further burden for programmers. In our work, we seek to deal with this situation by building automatic code translation tools that can generate optimized code to hand to each vendor's compiler from a single, specially written and annotated, but nevertheless standard Fortran source.

We have implemented our computational fluid dynamics (CFD) applications following this strategy, and we have successfully achieved high percentages of peak performance for multiple architectures. From this experience, we have identified SIMD engines as the key component in modern CPU and GPU designs that needs to be exploited to

* Corresponding author. Tel.: +1-612-626-1765.

E-mail address: phlin@cs.umn.edu.

obtain high performance. We find that coding to these SIMD components while also exploiting available on-chip data storage significantly improves computational efficiency. Based upon our experience with the Cell processor, we have developed a code optimization strategy to effectively reduce the memory footprint in the on-chip cache, implement pipelined computation to overlap data access with computation, and generate code written expressly for the SIMD engine. We find the automation of these optimizations feasible and have developed a translation tool to assist our code development for multiple platforms. Our performance results have shown the translated codes can deliver a high percentage of peak performance on the latest CPUs. From our team's own PPM gas dynamics codes, we have extracted two code modules, and we have then built each of these modules into a full application for testing. We have chosen these two test application codes so that they exhibit different types of calculations, each of which is representative of a portion of the work that would be done in a full CFD application. These test applications are: (1) PPM advection, (2) PPM single-fluid gas dynamics for low-Mach-number flows. In this paper, we use these two examples to present a study of performance portability covering both CPU and GPU devices. We evaluate GPU performance through manual translation to the CUDA language. Our results for the GPU are promising and compare well with the best GPU results for CFD applications of which we are aware.

2. Performance Challenges and Transformation Strategies

Exploiting the SIMD engines and the on-chip memory is the key to high performance computation on the Cell processor. We believe these same considerations are key to achieving high performance on multicore CPUs and GPUs. For efficient usage of these hardware features, the following performance challenges have to be tackled. First, to have efficient usage of the precious on-chip memory space, we must reduce the memory footprint and reuse cache-resident data. Second, to exploit the SIMD engines, we must ensure that the compiler generates SIMD instructions, or we must write programs using the SIMD intrinsic functions provided by the vendors. Finally, to overlap off-chip data accesses with computation, we must aggressively prefetch this data. This is easily done on the Cell processor, but on CPUs today, we can only give hints to the vendor compilers about this prefetching.

Our methodology for delivering uniformly high performance on multiple architectural platforms includes the following three steps. First, the programmer generates source code following a template structure. The code template used in this study is a cache-blocked code expression that updates “grid pencils.” A grid pencil is a sequence of contiguous “grid briquettes,” where each briquette is a small grid cube 2, 4, or 8 cells on a side. The briquette size is determined by the available on-chip memory size and the SIMD processing width. With briquette data records aligned on quad-word boundaries for computational efficiency, the grid planes of the briquette, with 4, 16, or 64 data values each, serve as SIMD operands. Our code transformation tool performs source code transformations to generate optimized and architecture-specific code. The first transformation focuses on the on-chip memory optimization. All relevant subroutines are first inlined. A grid plane packed with multiple physical state variables will be fetched from the main memory into the on-chip memory. These variables are unpacked into multiple temporary arrays and used for the data update. The code is then pipelined and the computation traverses each grid plane only once. This aggressive pipelining transformation has been described in detail elsewhere (see [1-2]). Data updates are executed via many vector floating point operations. We have to go through another aggressive memory reduction to ensure that the required data can fit into the on-chip memory. We perform liveness analysis in the code transformation and re-index the temporaries to assure that only the minimal set of transverse grid planes needed to compute a single updated plane are contained in the workspace. This transformation results in a huge reduction of the memory footprint in the on-chip cache. The second transformation generates code using the SIMD intrinsic functions to enforce vectorization for the SIMD engine. This process is necessary for the Cell processor,

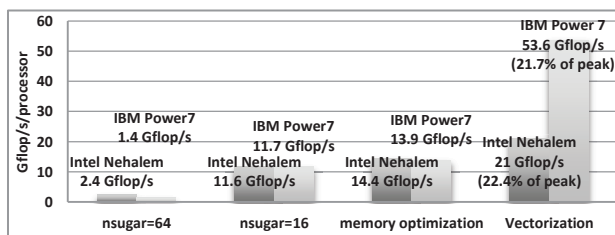


Figure 1: PPM advection performance on CPUs

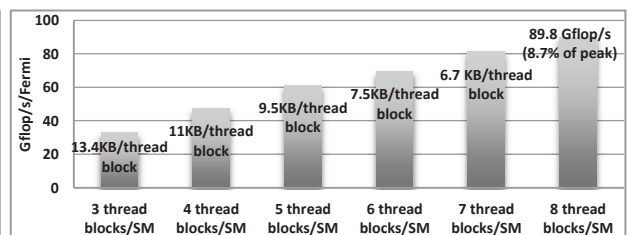


Figure 2: PPM advection performance on GPU

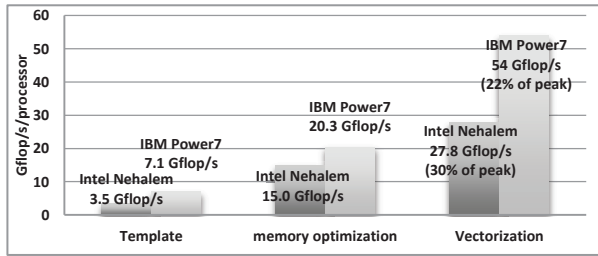


Figure 3: PPM single-fluid performance on CPUs

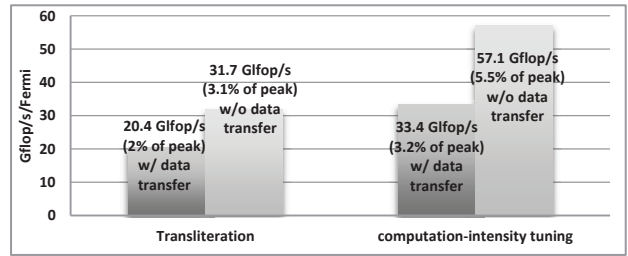


Figure 4: PPM single-fluid performance on Fermi GPU

and is optional but helpful for the CPU platform. CPU compilers provide similar intrinsic functions to support SIMD execution. To automate these code transformations, we use ANTLR as the infrastructure for the development of the transformation tool. Our tool for the on-chip memory optimization generates the memory optimized, and heavily pipelined code written in the Fortran language. Our vectorization tool then generates the SIMD code for different processors. The output is code written in one of multiple languages with the SIMD intrinsic extensions. Finally, the optimized code is submitted to the vendor compiler to produce a high performance executable.

3. Performance Test Results

Using multiple Cell processors in the large IBM Roadrunner system, we have achieved a performance between 3.4 Gflop/s/SPU (with thousands of Cell processors) and 3.9 Gflop/s/SPU (with hundreds of Cell processors.) Here, we present an extended performance study of the Intel and IBM multi-core processors and of the Nvidia GPU.

For the performance study of PPM advection, we begin with code that follows the code template regulations. The computation is expressed in a sequence of triply nested, vectorizable loops extending over a regular 3-D Cartesian grid. An outer code performs this computation on a sequence of grid briquettes of the whole, with the number of grid cells, *nsugar*, on each side of the cube given as a parameter. The leftmost group in fig. 1 shows the result for *nsugar* = 64; we obtain more or less the performance we would expect if we were to assign each briquette, with no cache blocking, to a separate MPI process, with MPI message passing occurring through on-node data copies in the shared memory of a single workstation. In this implementation the entire update does not fit into the on-chip cache, and the performance is low. Both the Intel Nehalem and IBM Power-7 CPUs show low performance in this case. Next, we perform our memory optimization using our translation tool. We process briquettes in sequence, but with *nsugar* = 4, and we pipeline the processing completely. During the time that we update a grid briquette entirely using cache resident data, we are pre-fetching the next briquette in the sequence. We also utilize partial results of one briquette update to perform the next briquette update, so that there is no redundant computation between the two updates. Comparing the results shown in fig. 1 for both processors, about 25% of the performance improvement is delivered by the cache memory optimization. The last case we study is the result from our vectorization tool applied to the memory-optimized code. The rightmost group in fig. 1 shows the transformed codes delivering the best percentage, 22.4% of 32-bit peak performance on the Intel CPU and 21.7% on Power-7. Thus both processors deliver a similar percentage of peak performance after our code transformations, although Power-7 has twice as many cores. To implement the advection code for the GPU, we transliterate the memory-optimized code into CUDA. We set the briquette size to 4^3 cells and use 64 GPU threads in a thread block to update a grid pencil. More than 13 KB shared memory is used per thread block, so that only 3 thread blocks are allowed in a single SM. The delivered performance is only 33 Gflop/s on a Fermi GPU (leftmost histogram in fig. 2).

The second performance test uses our PPM single-fluid code. We start the experiment with a grid of 128^3 cells. The computation is parallelized with 8 OpenMP threads and a brick of 64^3 cells is assigned to an individual thread. The performance is limited by the available memory bandwidth and only low percentages of the 32-bit peak performance are achieved. After on-chip memory optimization and computation pipelining, the optimized code has $3\times$ (Power-7) to $4\times$ (Intel) speedups in performance (center histogram in fig. 3). The SIMD codes generated from our transformation tool gain another $2\times$ speedup to deliver the 27.8 Gflop/s (30% of 32-bit peak performance) on the Intel CPU and 54 Gflop/s (22%) on the Power-7 (right histogram in fig. 3). Here, the 24.75 KB required on-chip memory space for a CPU thread shows the challenge in the GPU implementation. Under the expectation that shared memory cannot host the temporary results, we transliterate this example without any source code transformation into CUDA. This keeps all the temporary results in the global memory. The delivered memory throughput is close to the

available memory bandwidth and the achieved performance is 20 Gflop/s (left histogram in fig. 4).

To achieve better performance for the GPU, additional optimization should be applied to efficiently exploit the shared memory. For the PPM advection code, we have to reduce the shared memory usage. In this process, temporary variables that can be re-computed on the fly will be eliminated from the shared memory space. Through iterations of memory reduction, the best-delivered performance becomes 90 Gflop/s, which is 8.7% of the 32-bit peak performance (rightmost histogram in fig. 3). To exploit the shared memory for the PPM single-fluid code, we minimize the number of temporary results by de-coupling the equations and dividing the program into several episodes of computation. Each episode is like a subroutine and is called from a main computational kernel. In our GPU implementation, we only generate few GPU kernels, each performing a series of episodes of computation. This effectively eliminates the overhead caused by kernel launching and terminating. A single episode walks through the grid pencil to generate results used in the later episodes, or the final results. The inputs and outputs in an episode will have the size of a full grid pencil and be stored in the global memory. Each thread block reserves a private space in the global memory to store a group of pencil-sized arrays. A thread block can reuse the same group of arrays to perform multiple pencil sweeps. Intermediate results generated within an episode are cached in the shared memory. Because of the limited on-chip memory size, each episode is designed with only a handful of input and output variables. Instead of having high computational intensity, the intensity is controlled at around 5 flops per off-chip word for each episode. Only after this additional work in changing the numerical algorithm and reorganizing the code, we find our memory optimization is beneficial to this GPU implementation. This results in performance of 33.4 Gflop/s (with data transfer) and 57.1 Gflop/s (without data transfer), as shown in fig. 4.

4. Conclusion

In this paper, we present a study of performance portability covering the latest CPUs and GPU. Motivated by our experience with the IBM Cell processor, an optimization strategy and a code translation tool have been developed to assist code generation for high computational performance. A single set of code transformations works well for CPU devices. These transformations tackle different programming challenges and generate memory-optimized, and computation-pipelined codes written in different languages and using SIMD intrinsic functions. For our two PPM test applications on the CPU platforms, the delivered performance is 30% and 22% of the 32-bit peak. Transliteration into the CUDA language delivers relatively low performance. We have identified the challenges for our GPU implementations and provided solutions for the PPM applications. Applying those solutions manually, the best-delivered performances are 57 Gflop/s and 90 Gflop/s respectively. Although only a low percentage of peak performance is achieved on the GPU, these performance measurements are comparable to or higher than other CFD applications implemented on GPUs and reported in the literature cited here [3-4]. This paper has shown that our code transformation tool makes performance portability possible among the latest computational processors.

Acknowledgements

This work has been supported through grants CNS-0708822 and OCI-0832618 from the National Science Foundation and by the Department of Energy through a contract from the Los Alamos National Laboratory. We are also pleased to acknowledge helpful discussions on GPU programming with Guochun Shi at NCSA.

References

1. P. Woodward, J. Jayaraj, P. Lin, and P. Yew, "Moving scientific codes to multicore microprocessor CPUs," *Computing in Science & Engineering*, 2008, pp. 16-25.
2. P. R. Woodward, J. Jayaraj, P. Lin, and W. Dai, "First experience of compressible gas dynamics simulation on the Los Alamos roadrunner machine," *Concurrency and Computation: Practice & Experience*, vol. 21, no. 17, Dec. 2009, pp. 2160-2175.
3. GPU Acceleration of NWP: Benchmark Kernels Webpage. <http://www.mmm.ucar.edu/wrf/WG2/GPU/>
4. T. Shimokawabe et al., "An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code," in *SC International Conference for High Performance Computing, Networking, Storage & Analysis*, 2010, pp. 1-11